Bounding memory access interferences on the Kalray MPPA3 compute cluster

Dumitru Potop-Butucaru, Jad Khatib, Philippe Baufreton

Bounding memory access interferences on the Kalray MPPA3 compute cluster

Jad Khatib (Inria jad.khatib@inria.fr), Dumitru Potop-Butucaru (Inria dumitru.potop@inria.fr), Philippe Baufreton (Safran philippe.baufreton@safrangroup.com)

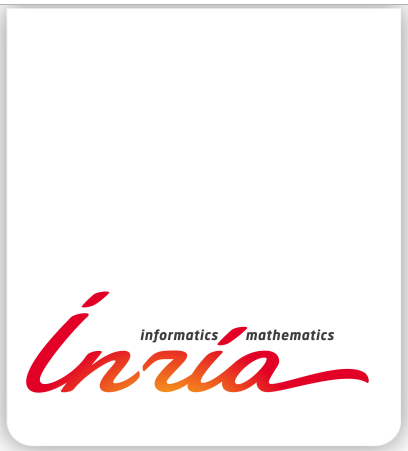
RESEARCH REPORT

N° 9404

December 2020

Project-Teams KAIROS

ISSN 0249-6399 ISRN INRIA/RR--9404--FR+ENG



Bounding memory access interferences on the Kalray MPPA3 compute cluster

Jad Khatib (Inria jad.khatib@inria.fr), Dumitru Potop-Butucaru (Inria dumitru.potop@inria.fr), Philippe Baufreton (Safran philippe.baufreton@safrangroup.com)

Project-Teams KAIROS

Research Report n° 9404 — December 2020 — 24 pages

Abstract: The Kalray MPPA3 Coolidge many-core processor is one of the few off-the-shelf high-performance processors amenable to full-fledged static timing analysis. And yet, even on this processor, providing *tight* execution time upper bounds may prove difficult. In this paper, we consider the sub-problem of bounding the timing overhead due to memory access interferences inside one MPPA3 shared memory compute cluster. This includes interferences between computing cores and interferences between the instruction and data accesses of a given core. We start with a detailed analysis of the MPPA3 compute cluster, with emphasis on three key components: the Prefetch Buffer (PFB), which performs speculative instruction loads, the fixed-priority (FP) arbiter between instruction and data accesses of a core, whose behavior is highly dependent (in the worst case) on interferences from other cores, and the SAP (bursty Round Robin) arbiters guarding access to memory banks. We provide a full-fledged interference analysis covering both levels. This analysis is rooted in a novel modeling of memory access patterns, which describes their worst-case and best-case burstiness, a key factor influencing the MPPA3 arbitration. We evaluate our interference model on multiple applications, ranging from real-life avionics code specified in SCADE to linear algebra code. We also suggests methods for reducing execution time and improving analysis precision by means of code generation.

Key-words: Real-time systems, Timing analysis, Interference analysis, Many-core, Kalray

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Analyse d'interférences mémoires sur les clusters de calcul du pluri-cœurs Kalray MPPA3

Résumé : Le pluri-cœurs Kalray MPPA3 Coolidge est un des seuls processeurs haute-performance sur étagère à permettre le calcul de bornes statiques (non-probabilistes) sur le temps d'exécution. Mais même sur ce processeur le calcul de bornes serrées est difficile. Dans cet article, nous traitons le sous-problème du calcul de bornes supérieures sur les interférences dues aux accès concurrents aux bancs de mémoire partagée. De plus, notre analyse se concentre sur un seul cluster de calcul de l'architecture-cible, et s'intéresse seulement aux interférences entre cœurs de calcul du cluster et aux interférences entre accès instruction et données d'un seul cœur. Nous commençns par une analyse détaillée du cluster de calcul MPPA3, mettant l'accent sur trois composants-clefs: le tampon de préchargement anticipé (Prefetch Buffer, ou PFB) qui réalise des préchargements de code spéculatifs, l'arbitre à priorité fixe (FP) entre les accès au code et aux données d'un même cœur de calcul, dont le comportement est dépendant (au pire cas) des interférences d'autres cœurs, et les arbitres SAP (Round Robin avec support pour les rafales) qui contrôlent l'accès aux bancs de mémoire partagée. Nous développons une analyse d'interférences complète par rapport au domaine choisi. Notre analyse est fondée sur une nouvelle modélisation des motifs d'accès à la mémoire, qui permet la représentation du groupage des accès en rafales (dans le pire et dans le meilleur des cas). Ce facteur a une influence très forte sur l'arbitrage MPPA. Nous évaluons notre approche d'analyse d'interférences sur plusieurs applications allant de tâches avioniques appartenant à une application de production spécifiée en SCADE, et jusqu'à du code d'algèbre linéaire représentatif pour les applications de type "jumeau numérique" ou "machine learning". Nous suggérons aussi des méthodes permettant de réduire le temps d'exécution et d'améliorer la précision de l'analyse par des choix de génération de code.

Mots-clés : Temps réel, Analyse de temps d'exécution, Analyse d'interférences, Pluri-coeur, Kalray

1 Introduction

The world of embedded computing is rapidly changing. The classical embedded control system, with its (relatively) low computational requirements is progressively extended to include AI/ML components or model predictive control[1] with high computational needs (and tight real-time requirements). Implementing such systems requires methods and tools belonging to not one, but two major scientific and engineering fields:

- Real-Time Embedded (RT/E) computing, for the aspects related to safety and predictability.
- High-Performance Computing (HPC), for the aspects related to performance and efficiency.

The Kalray MPPA family of many-core processors[2] is one of the most promising results of this on-going synergy between RT/E and HPC computing. It has been developed as a hardware solution meant to provide support for both raw performance needs and predictability. Raw performance is attained by means of massive parallelism[3] and carefully designed memory system, on-chip interconnect, and I/O interfaces. Attaining predictability usually requires both transparency—access to the processor specifications to allow analysis—and avoiding as much as possible mechanisms that are known to reduce the precision of analysis [3], such as out-of-order pipelines, speculation, non-LRU caches... Kalray does both [6, 10].

However, even this level of hardware support for predictability does not make timing analysis easy. While the in-order VLIW pipeline of each MPPA3 processing code is amenable to very precise timing analysis, two fundamental issues remain:

- Attaining both performance and predictability is difficult on any multi- and many-core processor [4], as performance requires some degree of resource sharing,[4] whereas predictability is traditionally attained through time/space isolation mechanisms meant to eliminate interferences [2], or at least significantly reduce and bound them.[5]
- The quest for raw performance means that, even though Kalray MPPA processors are the best for predictability among production HPC-capable processors, it still features hardware components that make analysis difficult and reduce its precision. On the MPPA3, these components perform *speculative* memory fetches, L1 *fixed-priority (FP) arbitration*[6] between code and data memory requests of a given processor core, *bursty L2 and L3 arbitration with bounded (not fixed) burst size* between memory requests coming from different cores/clusters, or (more classically) they access DDR memory whose refresh operations must be considered during timing analysis.

The first topic is covered in ongoing work which has already produced a few interesting compromises [16, 6, 7]. **Our goal, in this paper, is to focus on the second topic,** and more precisely on the issues related to executing parallel code on a **single compute cluster** of the MPPA3 processor.

Our contribution is threefold:

[1] Digital twins in the control loop.

[2] `www.kalrayinc.com`

[3] Large numbers of efficient computing pipelines and specialized accelerators.

[4] For instance, allowing two or more processors to access the same memory bank concurrently.

[5] We shall not consider in this paper mixed-critical solutions where the predictability of high-performance parts of an application is attained by means of monitoring and scheduling.

[6] Which is dependent on the behavior of lower levels of the memory hierarchy.

- We conduct an analysis of the Kalray MPPA3 compute cluster, focusing on the memory subsystem and on the three components that pose static analysis problems (prefetch buffer, L1 arbiter and L2 arbiter). We identify worst-case interference patterns which result in better interference bounds than over-approximations of previous work. In doing so, we determine the importance of memory access burstiness in the definition of these patterns, and we also identify methods to reduce interferences by construction.
- We introduce a formal model allowing the representation and worst-case reasoning on bursty memory access pattern specifications. On this base we develop novel algorithms allowing to derive worst-case upper bounds on the memory access interferences at both L1 level (between instruction and data requests of the same core) and L2 level (between requests coming from different cores).
- We evaluate our model and algorithms on multiple applications: a classical embedded control application (a real-life avionics software specified in SCADE), a few applications of the TACLE WCET testbench [8], and a linear algebra code (matrix multiplication) which is representative for high-performance AI/ML code.

Our work was guided by two main objectives: precision and scalability. To attain precision, we consider in great detail the properties of the hardware. To attain scalability, we have systematically avoided HW state exploration approaches, relying instead on static over-approximations.

Outline The remainder of the paper is organized as follows: In Section 2 we review previous work. Section 3 presents in detail the MPPA3 architecture, with focus on the compute clusters. In Section 4 we set the basis for the definition of the interference model by formally defining the architecture and application model and by setting the general timing analysis paradigm we follow. Sections 5 and 6 defined our methods for L1, respectively L2 interference analysis. Section 7 covers experimental evaluation, Section 8 discusses methods for reducing interferences, and Section 9 concludes.

2 Related Work

Our work is closely related to the large corpus of previous results on the timing analysis of parallel code running on multi-cores, regardless of whether analysis is considered as an end in itself [11, 13], or if it is seen as part of a larger ressource allocation process meant to provide hard guarantees on the real-time behavior of the resulting code [15, 6, 16, 12, 17, 7]

One main difference between our approach and the ones cites above is that we do not consider the full-fledged timing analysis problem. Instead, given that multiple integrative approaches exist (cited above), we only focus on providing upper bounds on L1 and L2 memory access interferences.

In doing this, we follow a *multicore response time analysis*[4] approach where, as detailed in Section 4.2, the worst-case respose time (WCRT) of a task is computed as a sum between a worst-case execution time (WCET) assuming no interferences plus separate terms corresponding to the various interference sources (in our case, L1 and L2 interferences). In particular, we assume that WCET analysis does not even consider the state of the L1 arbiter (where data and instruction memory accesses of the same PE interfere). Theoretically, considering the state of this arbiter during WCET analysis could significantly improve analysis precision. However, the presence of speculation and the dependence of L1 arbitration on the behavior of lower levels of the memory hierarchy means that the state space to explore is very large, leading to tractability issues which we wanted to avoid.

Much of the previous work cited above has (also) been has targeted at the Kalray MPPA platform, and in particular its MPPA2 version (codenamed Bostan). Our paper paper considers the hardware innovations brought by the Kalray MPPA3 platform—speculative prefetch and bursty arbitration–which have only been considered in [10]. By comparison, we provide significantly tighter bounds on the L1 interference analysis and the first correct L2 interference analysis, both based on a novel modeling of burstiness which we introduce.

3 The Kalray MPPA3 many-core processor

The third generation of Kalray many-core processors (codenamed Coolige) integrates 80 Processing Elements (PEs) running at 1200 MHz, distributed over 5 identical Compute Clusters (CCs). CCs are interconnected through a dual Network on Chip (NoC) and through an Advanced eXtensible Interface (AXI). The AXI network also connects the CCs to two DDR controllers (each guarding access to 4GB of RAM each), and to a PCIe controller. The NoC also connects the CCs to two 100G Ethernet controllers.

In this paper, we focus our study on a single CC. Thus, we do not consider in our analysis the NoC and the AXI interconnect.

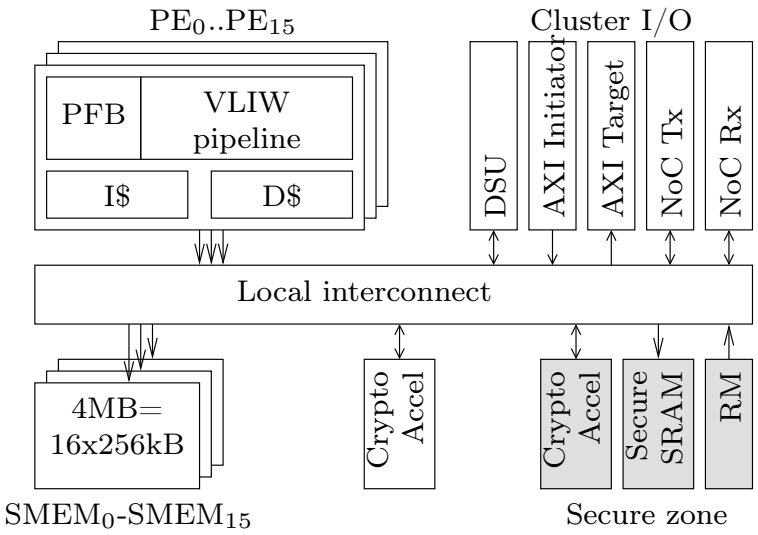


Figure 1: MPPA3 Compute Cluster

3.1 Compute Cluster

The overall organization of a CC is depicted in Fig. 1. It consists of two interconnected zones: a secure one and a non-secure one. The secure zone contains a Resource Management (RM) processor, a 256 KB secure memory bank and a dedicated cryptographic accelerator. The non-secure zone contains:

- 16 identical processing elements (PEs)
- 16 identical shared Static RAM banks of 256 KB each, totaling 4 MB of shared local memory (SMEM)

- Two Direct Memory Access (DMA) interfaces allowing receiving (Rx) and transmitting (Tx) data over the NoC
- An AXI channel interface consisting in two components, one initiator for receiving requests from outside the CC, and one target issuing requests to the other CCs and I/O devices.
- A second cryptographic accelerator and a Debug Support Unit (DSU).
- A real-time clock is associated with each PE, and clocks in the same cluster are synchronous.

In this paper, we focus on the interaction between the 16 PEs and the 16 SMEM banks of the non-secure zone, which is critical in attaining performance.

3.2 Processing element (PE)

As pictured in Fig. 1 and Fig. 2(left), each PE consists of a 64-bit 6-issue Very Long Instruction Word (VLIW) pipeline alimented by one prefetch buffer (PFB). The PFB is connected to the L1 instruction cache and the pipeline is directly connected to the L1 data cache. Instruction and data requests towards the memory are multiplexed using a fixed priority (FP) arbiter that gives priority to data requests (either read or write).

The VLIW pipeline issues *bundles* formed of one or more *instructions*. Each instruction is formed of 1 to 3 32-bit words, called *syllables*, and a bundle can contain 1 to 8 syllables.[7] The VLIW pipeline is timing compositional [5], meaning that execution is monotonic: shorter functional unit durations and faster responses from memory result in shorter execution time.

3.2.1 Prefetch bufer (PFB)

While the VLIW pipeline is timing compositional, the pipeline is alimented by a prefetch buffer which performs speculative prefetching. It has 4 lines of 4 syllables each, and it always attempts to keep the lines filled with code. It does so by requesting instructions from the instruction cache (one PFB line at a time) each time the PFB contains free lines.

When execution reaches and takes a jump operation—unconditional jump, conditional jump, call, or hardware loop end—the PFB is flushed and the prefetch process starts from scratch.

Speculation means that the number of memory requests issued by the PFB to the cache (and thus to the memory) may vary, even for the same code executed on the same data. Indeed, faster instruction loads from memory due to reduced interferences from other cores may result in the PFB issuing more requests before a jump. In turn, this may result in a longer overall execution—a timing anomaly [14].

Note that each point where execution reaches a jump operation may be subject to speculative loading of at most 4 PFB lines of code that may not be executed. This means that effects can accumulate in time, but also that this effect is bounded. We shall always, in our analyses, assume a worst case where the PFB manages to fill in the PFB lines before each jump that is taken. This results in at most 4 requests to the instruction cache and 2 requests to memory, each of which is potentially subject to interferences. However, it is important to note that *only the last of these two speculative memory requests may delay pipeline execution and thus increase execution time.* This is because the last request is issued just before the PFB is flushed in response to the jump being taken, and thus delays the load of the first bundle after the jump.

Due to speculative prefetching, the precision of the analysis will be reduced for code featuring many jump instructions that are taken. To avoid creating fine-grain branching, the Kalray3

[7] Grouping instruction into bundles is subject to complex functional constraints. For instance, two memory operations cannot be part of the same bundle.

instruction set (KIS) provides a few *predicated* operations [9], such as conditional move operations or conditional load/store operations. These operations take as input a predicate register, and the operation is executed only if the predicate is non-zero. This allows conditional execution without branching, and thus without the associated PFB-related imprecision.

3.2.2 L1 caches and memory request sizes

Both L1 caches are 4-way set-associative with Least Recently Used (LRU) replacement policy, a 64 byte (16 syllables) cache line and 16 Kbytes total size.

The data cache is write-through, with a no-write-allocate write policy. This means that write operations are forwarded synchronously to the memory and to the cache itself. If the data is in cache, the cache state is updated. If not, the cache state remains unchanged.

The bus between caches and memory is 256 bits wide (32 bytes, 8 syllables). Each memory request can transport at most this amount of data, and thus is issued in exactly 1 clock cycle.[8] This means that loading a cache line from memory requires 2 read requests issued without a gap between them (collated).

Write requests can only be issued for data (by the Load/Store unit of the pipeline). A write request only takes 1 cycle, and corresponds to exactly one store operation of the program. In particular, store requests are not grouped to reduce the number of memory accesses, meaning that an SB or SW operation (store byte/word) operation will only use 8, respectively 32 bits of the 256 bits of the memory bus width. The KIS provides store operations all the way from from byte size to octuple word size (256 bits) and the compiler attempts to group smaller store operations into larger ones.[9] However, for typical software (like those in our test bench), the reduction obtained by automated grouping is not significant, so that large numbers of write requests will be issued to memory. This also means that aggressive program optimization, which reduces the number of memory accesses by working as much as possible in the processor registers, is key in reducing interferences.

After a read request resulting in a miss, the PFB or pipeline cannot issue a new (read or write) request to memory until the missing cache line is retrieved from memory.

Cache coherency between PE caches can be enabled, but as our study is mainly concerned with predictability, and given that cache coherency makes timing analysis more difficult, we assume that cache coherency is disabled.

3.3 Local cluster memory (SMEM) and local interconnect

Each CC contains 4Mbytes of Static RAM accessible to all PEs.[10] This Shared MEMory (SMEM) is partitioned into 16 banks of 256Kbytes each, which we denote $SMEM_0 - SMEM_{15}$.

The SMEM can be used under 3 configurations:

Banked mode This is the classical memory space organization, where each SMEM bank is assigned a contiguous physical address range. This mode allows maximum control over how data and code are allocated to specific banks, allowing the enforcement of space isolation rules that may significantly (or totally) reduce interferences between cores.

[8] The mechanism of the previous-version Kalray 2, where *packets* containing multiple *flits* allow transporting more data than the bus width is not used. Instead, the bus is significantly wider, and burstiness support is added to the arbitration.

[9] Stardard library routines, such as `memcpy` can also be (manually) optimized, which results in significant gains in programs that use them, such as code generated from Lustre/SCADE.

[10] In addition, it also has a secure 256Kbytes bank, only accessible to the RM PE.

Interleaved mode Consecutive physical memory locations are assigned in consecutive (modulo 16) memory banks. Under uncontrolled memory access patterns, this has a load balancing effect, usually reducing average-case interferences. However, space isolation approaches become virtually impossible, which largely complicates worst-case static interference analysis.

L2 cache mode All or part of the local memory can be configured as L2 cache. Non-partitioned shared caches are a difficult topic in timing analysis, because their states are difficult to predict, making this approach incompatible with our objective of timing predictability.

For the remainder of the paper, we shall assume that the SMEM is configured in banked mode. The *local interconnect* of the CC ensures that:

- Accesses from two different PEs to two different SMEM banks do not interfere with each other.
- In the absence of interferences, the memory access time does not depend on the PE issuing a request and the target SMEM bank (uniform memory access model).

In the absence of interferences, the memory pipeline latency[11] is of 23 cycles.

3.4 Memory access arbitration

The structure of the memory system of the MPPA3 processor is pictured in Fig. 2. We can distinguish 3 arbitration levels:

L1 arbitration happens inside a PE between the data and instruction requests going towards the memory. It is done by a single fixed priority (FP) arbiter.

L2 arbitration happens inside a CC. The access to each SMEM bank and to the AXI target is guarded by one arbiter with SAP policy (detailed below). These arbiters receive requests from the PEs and from the other components of the CC (RM, DSU, accelerators...).

L3 arbitration happens inside the AXI interconnect. One AXI arbiter with DRR policy guards access to each of the two DDR memory controller and to each of the CCs.

Note that, through AXI, a PE of one cluster can access not only the external DDR memory, but also the SRAM of other CCs.

L1 arbitration is of fixed priority (FP) type, with priority being given to data accesses. L2 arbitration follows a modified, configurable Round Robin policy called Smart Arbitration Policy (SAP). Under SAP, when a source of requests (such as a PE) is granted access, $(n+1)$ successive requests will be accepted if they come in successive cycles starting on the cycle where access is granted. Here, n is a per-CC configuration parameter that takes a value in the range 1..7.

For space reasons, we focus our analysis on the *intra-cluster* L1 and L2 arbitration and on the interaction between PEs and SRAM banks of memory, assuming that other components (grayed in Fig. **??**) do not contribute to the interferences. This amounts to assuming that software organization and synchronization ensures the absence of interferences from these sources. Extending our analysis to include the L3/AXI level and interferences from the other sources is ongoing work.

[11] Time duration between a cache issuing a memory read request upto the point where the memory respose arrives back to the cache.

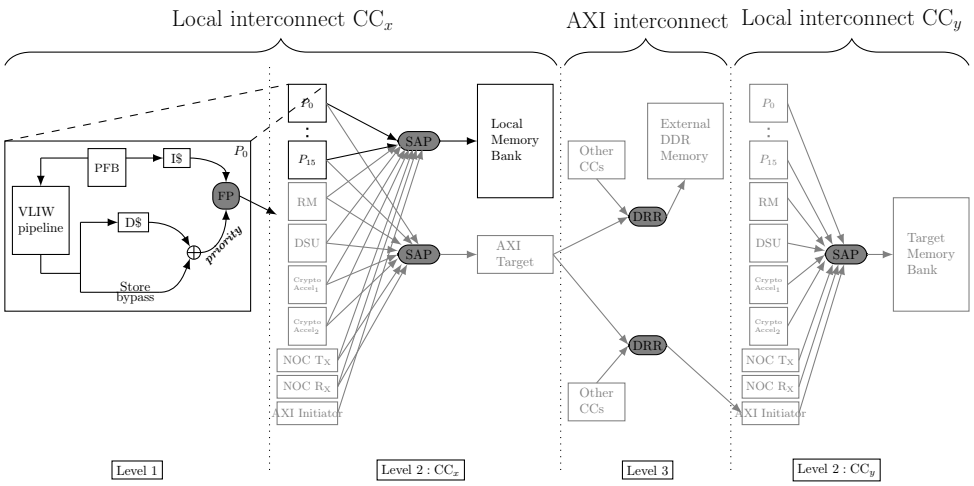


Figure 2: Memory system of the MPPA3 processor

4 Interference analysis method overview

In this section, we first formalize the application and architecture model, and then define the general timing analysis paradigm we follow.

4.1 Application and architecture model

We assume the application code we analyze uses $1 \leq P \leq 16$ processing elements and $1 \leq B \leq 16$ memory banks out of those provided by the CC. We label the PEs we use $p_0 - p_{P-1}$, and we label the memory banks $b_0 - b_{B-1}$. Note that p_0 and b_0 do not necessarily correspond to the PE_0 or SMEM_0. Indeed, in common configurations of the MPPA3 cluster, the first 2 of the 16 banks of the CC are used by system software, so that b_0 cannot correspond to them.

We assume that $p_0 \dots p_{P-1}$ and $b_0 \dots b_{B-1}$ are fully dedicated to the execution of the application code we analyze. In particular, no accesses from other sources (other PEs or other I/O devices) target $b_0 \dots b_{B-1}$ in the analysis timeframe, and the $p_0 \dots p_{P-1}$ only execute code of the aplication under analysis which only access memory banks $b_0 \dots b_{B-1}$.

The application under analysis consists in a set of *non-preemptive* tasks t_i $0 \leq i \leq (T-1)$. We assume the allocation of tasks to PEs is fixed. We denote with $p(t)$ the execution PE of task t.

To focus on the low-level arbitration aspects that interest us, we assume that the potential interference matrix is provided. More precisely, for every two tasks t_i and t_j with $i \neq j$ a Boolean $overlap(t_i, t_j)$ determines whether the two tasks can overlap in time, and thus interfere. Various methods have been proposed in the literature for determining whether two tasks can overlap/interfere, for both dependent/DAG task models and time-triggered task models.

We assume tasks do not perform uncached data memory reads. Thus, the execution of a task will result in only 3 types of memory accesses: cached instruction memory reads, cached data memory reads, and uncached data memory writes.

To facilitate presentation, we assume that the execution of each task starts with an empty pipeline and empty caches. This state can be attained using `barrier` instructions.

4.2 Response time analysis

We assume that for each task t one can compute $WCET(t)$ which is a safe upper bound on the execution duration of t in isolation (without L2 interferences from other tasks) while assuming that the PE executing t has not one access to memory (subject to FP arbitration) but two (separate for code and data), and also assuming that code and data are placed on separate memory banks, so that no L1 or L2 interferences exist between code and data traffic generated by t.

Note that $WCET(t)$, as defined here, is not a true upper bound for the actual execution of code on a MPPA3 PE, even assuming execution in isolation.

On the MPPA2 platform, L1 interferences were considered as part of the WCET value. However, two innovations of the MPPA3—the introduction of the speculative prefetch buffer (PFB) and the fact that L1 arbitration depends on L2 interferences[12]— significantly decouple pipeline execution from the L1 arbitration, which explains our choice to separate them. Note, however, that the analysis producing $WCET(t)$ must still consider PFB-related and cache-related contributions.

We denote with $WCRT(t)$ an upper bound on the execution of task t, which also takes into account the L1 and L2 interferences. Under a *multicore response time analysis*[4] approach, we consider different interference sources (L1 and L2) as separate factors, which gives the following formula for the response time of a task:

$$WCRT(t) = WCET(t) + interfL1(t) + interfL2(t)$$

where $interfL1(t)$ is an upper bound on the L1 interferences on the execution of t (a time overhead expressed in clock cycles) and $interfL2(t)$ is an upper bound on the L2 interferences on t by tasks t' that may overlap in time with t ($overlap(t, t') = true$). We assume that $overlap(t, t') = false$ for all task t' with $p(t') = p(t)$.

One important point here is that $WCET(t)$, $interfL1(t)$, and $interfL2(t)$ are independently computed. For instance, $interfL1(t)$ and $interfL2(t)$ may correspond to different execution scenarios.

5 L1 Interference Analysis

5.1 Worst-case single interference cost

Recall that the L1 arbiter is of fixed priority type (FP), giving priority to data traffic. The main problem related to the use of FP arbitration is the potential for starvation for the lower-priority traffic—a request may be indefinitely denied because high-priority request come without interruption.

In the MPPA3 PE, an instruction request (issued by the instruction cache) can be delayed by multiple data requests [10]. However, this delay is bounded:[13] an instruction request blocked at the level of the L1 arbiter blocks the instruction cache, and thus does not allow the load of new instructions by the PFB. Even assuming all instructions in flight in the pipeline and in the PFB are memory accesses, when these instructions are all completed the instruction request will pass L1 arbitration.

[12] During an execution in isolation, two memory accesses generated by non-consecutive bundles cannot traverse the L1 arbiter in successive cycles. However, when the memory pipeline is blocked at L2 level by requests from other processors, the L1 arbiter can be blocked itself, allowing new requests can "catch up" with the ones blocked at L1 level.

[13] A phenomenon known as *bounded starvation*.

To determine the worst case scenario, recall that a load operation resulting in a data cache miss will block the data cache until a response is received from the memory. Thus, an instruction request can always pass just after a read request sent by the data cache to the memory.

The PFB can store up to 16 bundles formed of one load or store instruction each, and the pipeline can contain 4 in-flight store operations. Then, the maximum delay L1 arbitration can inflict to a instruction request happens when the pipeline and PFB contain a sequence of 19 store operations followed by one load operation resulting in a data cache miss. As each store operation takes one cycle at the L1 arbiter level and each cached load operation takes 2 cycles, the maximal total delay is $d = 21$ cycles.

5.2 Worst-case number of interferences

Every instruction read issued by the instruction cache is potentially subject to an interference. However, determining the number of such operations is not straightforward in the presence of the speculative PFB. To up-bound the number of accesses, we can make the worst-case assumption that before each jump[14] that is taken the PFB has the time to fill up. Given that the PFB size equals that of 2 cache lines (but with potentially unaligned accesses) this makes for at most 2 instruction cache requests corresponding to speculative loads that are not used.

However, among these two unused speculative loads only the last one may delay execution—the previous one is completed before the jump instruction is performed.

Existing static analysis tools such as aiT[15] or OTAWA[1] already can determine, in addition to the task WCET, an upper bound on the number of read requests issued by an LRU instruction cache when prefetch units are not present. Their analysis can be extended to include the worst-case PFB request scenario detailed above, and thus to produce an upper bound on the number of read operations issued by the instruction cache where interferences result in execution time delays.

We denote with $icache\_req(t)$ this number of requests.

5.3 L1 interferences - coarse upper bounds

Worst-case single interference cost d and the upper bound on the number of interfering instruction cache requests $icache\_req(t)$ provide us with a first upper bound on the L1 interferences:

$$interfL1^0(t) = d * icache\_req(t)$$

This first (and coarsest) upper bound has already been introduced in previous work [10].

We denote with $w(t)$ an upper bound on the number of data write requests issued during the execution of t and with $r(t)$ an upper bound on the number of data cache misses during the execution of t.[16] Recall, from Section 3.2.2, that each write request takes 1 cycle on the L1 arbiter and each data cache miss results in two collated read requests taking 2 cycles on the L1 arbiter. Then, the L1 interferences associated with t are bounded by $w(t) + 2 * r(t)$, and we can refine the previous bound into:

$$interfL1^1(t) = min(w(t) + 2 * r(t), interfL1^0(t))$$

[14] Unconditional jump, conditional jump, or hardware loop iteration.

[15] `www.absint.com/ait/`

[16] Both $w(t)$ and $r(t)$ can be computed using existing WCET analysis tools.

5.4 Burstiness and refined upper bound

In the the definition of $interfL1^0(t)$ and $interfL1^1(t)$, the $icache\_req(t)$ term is difficult to optimize, as prefetching decorrelates instruction reading from pipeline execution. However, systematically considering the penalty of d is an obvious over-approximation, as the worst-case scenario to which it corresponds, while feasible, is difficult to attain.

To understand how this penalty can be reduced, we need to go back to the presentation of Section 5.1, and understand how the sequence of data accesses that delays an instruction requests at L1 level is formed. The first remark is that such a sequence, or *burst*, of data accesses (with no free cycle between them) is formed of between 0 and 20 write requests followed by zero or two collated read requests (but never more than 21 requests total).

In every execution trace ϕ of t, we can count these bursts and classify them by size. The result is a function $e_\phi : \{1, \ldots, 21\} \to \mathbb{N}$ giving for each burst size i the number $e_\phi(i)$ of bursts of that size. We denote with $\mathbb{B}_{21}$ the set of such burst descriptions.

If $e_\phi \in \mathbb{B}_{21}$ is known, then an upper bound on the L1 interferences of trace ϕ is given by:

$$interfL1^2(t, e_\phi) = \sum_{k=21}^{1} k * min\left(e_\phi(k), \left\lceil icache\_req(t) - \sum_{l=21}^{k+1} e_\phi(l) \right\rceil\right)$$

This formula amounts to assuming that the larger bursts are causing interferences before smaller ones.

To allow moving from a per-trace formula to a trace-independent formula, we first introduce a partial order relation on $\mathbb{B}_{21}$. If $e^1, e^2 \in \mathbb{B}_{21}$ we say that e^2 dominates e^1, denoted $e^2 \geq e^1$ if for every $21 \geq k \geq 1$ we have:

$$\sum_{i=k}^{21} i \times e^2(i) \geq \sum_{i=k}^{21} i \times e^1(i)$$

This amounts to e^2 taking at least as many cycles at the L1 arbiter as e^1, and e^2 having these cycles grouped into greater bursts. The set $\mathbb{B}_{21}$, endowed with the $\leq$ partial order, is a lattice. We shall denote with $\vee$ the lower upper bound operator of this lattice.

Under these definitions, $e^2 \geq e^1$ implies $interfL1^2(t, e^2) \geq interfL1^2(t, e^1)$, and therefore a trace-independent upper bound on the L1 interferences is provided by:

$$interfL1^3(t) = interfL1^2(t, \bigvee_{\phi \ trace \ of \ t} e_\phi)$$

We denote $e(t) = \bigvee_{\phi \ trace \ of \ t} e_\phi$, and call it the L1 burst characterization of t. In fact any burst characterization of $\mathbb{B}_{21}$ that is greater than $e(t)$ will provide a safe upper bound for $interfL1(t)$.

Before moving on, we introduce two more notations. If $x \in \mathbb{B}_{21}$ we shall allow its representation as $(i_1 \to x(i_1); \ldots; i_m \to x(i_m))$, where $21 \geq i_1 > \ldots > i_m \geq 1$ are the indices where $x(i_l) \neq 0$. For instance, $(5 \to 2; 1 \to 5)$ defines a burst description composed of 2 bursts of size 5 and 5 bursts of size 1. By extension, () is the empty burst description containing no burst. On $\mathbb{B}_{21}$ we introduce the addition operation $+$, which is defined pointwise: $(a + b)(i) = a(i) + b(i)$ for $1 \leq i \leq 21$.

5.5 Tight over-approximation of e(t)

Our final objective at L1 level is to provide a method for computing a tight over-approximation of $e(t)$. To do so, we start by noting that, when the instruction cache isuses a read request towards the memory (at the request of the PFB):

- It can only be delayed by memory access instructions already in the pipeline and PFB that actually result in data memory requests (going beyond the data cache). All data write operations in the pipeline and PFB fall in this case, but only data read operations resulting in a miss must be considered. Furthermore, when a read is considered, it terminates the sequence of memory accesses that delays (the instruction request can pass after the data read request, as explained above).

- If the instruction read request towards memory occurs immediately after a branch statement that is taken (and which is accompanied by a PFB flush) and if the target instruction of the branch is not in the instruction cache,[17] then only the memory access instructions preceding the branch may delay the instruction read request.

For simplicity, and in order to focus on the arbitration-related issues, we shall assume that the control flow of our tasks does not involve loops (that it is a directed acyclic graph - DAG), and that it has a single input point and a single exit point.

The objective of our analysis is to determine, for each bundle p of task t, an L1 burst characterization of t, assuming that its execution ends in p. We denote this characterization $e(t,p)$. Then, we can set $e(t) = e(t, p_{end})$, where p_{end} is the exit bundle of t.

The definition of $e(t,p)$ is inductive and has four cases. The **first case** is when p is the first bundle of trace t. Then, we can set:

$$e(t,p) = \begin{cases} () & \text{if } p \text{ contains no memory access} \\ (1 \to 1) & \text{if } p \text{ contains a store instruction} \\ (2 \to 1) & \text{if } p \text{ contains a load instruction that may generate a cache miss} \end{cases}$$

The **second case** is when bundle p contains no memory access instruction that may result in a request being sent to memory. This means no store instruction and no load instruction that may result in a data cache miss.[18] In this case, we start by denoting $src(p)$ the non-void set of bundles that can directly give control to p (either in sequence or through branching instructions). Then, we can set:

$$e(t,p) = \bigvee_{p' \in src(p)} e(t,p')$$

The **third case** is when p contains a store instruction. In this case, we start by identifying all the sequences of bundles $p_k \dots p_1 p$ ending in p and having the following properties:

- $p_k \dots p_1 p$ can be part of a valid execution trace.

- The bundles $p_{k-1} \dots p_1 p$ can all be together in flight in the pipeline and the PFB.

- The sequence contains at most one load instruction that will certainly result in a data cache miss.[19] If present, this load instruction cannot be followed by other store instructions.

We denote this set of bundle sequences with $P(p)$. Given that pipeline and PFB can contain only 4 bundles and 16 syllables, this set is finite (and usually small). This set contains all the sequences of bundles whose memory accesses can be grouped in a burst ending in the write of bundle p, and which can delay a memory request of the instruction cache. Considering $w = p_k \dots p_1 p \in P(p)$,

[17] Which can be determined through a Must cache analysis.
[18] As determined by a Must cache analysis.
[19] As determined by a May cache analysis.

we shall denote with $size(k)$ the size of the maximal associated burst that can be generated by bundles $p_{k-1} \dots p_1 p$. Then, we can set:

$$e(t,p) = \bigvee_{p_k \dots p_1 p \in P(p)} (e(t,p_k) + (size(p_{k-1} \dots p_1 p) \to 1))$$

This formula amounts to exploring all possible decompositions of the potential memory accesses into bursts.

The **fourth case** is when p contains a load instruction that may result in a data cache miss. When the cache miss happens, the only burst ending in the read access contains the read itself. Thus, we can set:

$$e(t,p) = (\bigvee_{p' \in src(p)} e(t,p')) + (2 \to 1)$$

Note the reuse of the reasoning of the second case, and the addition of the burst of size 2, which corresponds to the two collated memory requests due to the data cache miss.

6 L2 Interference Analysis

Recall from Section 3.4 that L2 arbitration is performed using a modified Round Robin policy called SAP. The policy, which is configurable at CC level by a constant $1 \leq n \leq 7$, will allow each PE, when it is granted access, to pass at most $n+1$ requests, if they arrive in successive cycles. Therefore, burstiness is again key in analyzing arbitration. However, several key differences with respect to L1 arbitration require an extension to our modeling apparatus:

- Instruction and data traffic have already been mixed at L1 level (we need to consider both).
- Interferences happen at bank level, meaning that from the traffic produced by a specific PE we need to extract the component concerning a specific bank. Identifying bursts is not straightforward in this context.
- At L1 level, a worst-case analysis of the greatest bursts was sufficient. At L2 level, the worst case will be given by the greatest bursts produced by one core interfering with the smallest bursts produced by another.

For this reason, we start by extending our burst modeling apparatus.

6.1 Burstiness modeling, part 2

While at L1 level we were only interested in bursts of at most 21 data memory requests, we must consider here two different kinds of bursts:

- Bursts of successive memory requests of unbounded size issued by one PE to one memory bank. We denote the set of such burst descriptions $\mathbb{B}_\infty$.[20]
- Bursts of size at most $n+1$ accepted by the SAP arbiter from one PE. We denote this set $\mathbb{B}_{n+1}$

[20] Large bursts of this type can easily be generated in practice by long sequences of store instructions pre-loaded in the cache (e.g. as part of a loop).

The second type of burst descriptions are obtained by fragmenting the potentially unbounded bursts issued by the PE according to the SAP policy. This operation is performed using function $frag_n : \mathbb{B}_\infty \to \mathbb{B}_{n+1}$ defined by:

$$frag_n(b)(i) = \begin{cases} \sum_{\substack{1 \leq k \leq \infty \\ k \bmod (n+1) = i}} b(k) & \text{if } 1 \leq i < n+1 \\ \sum_{1 \leq k \leq \infty} (b(k) * (k \; div \; (n+1))) & \text{if } i = n+1 \end{cases}$$

Given that the $\mathbb{B}_{n+1}$ burst descriptions can thus be obtained from $\mathbb{B}_\infty$ descriptions, we shall assume that each task is only characterized in the $\mathbb{B}_\infty$ domain. Such characterizations must produced by static analysis methods similar to those of Section 5.5.

For each task t and for each memory bank b accessed by t we need to consider not one, but two burst descriptions:

- When determining how much t can delay another task t' through interferences happening on bank b, then we need a burst description denoted $coarse(t, b)$ that is greater, in the $\leq$ partial order of Section 5.4, than the burst description of any execution trace (as explained in Section 5.4) of t in the system including t and t'.
- When determining how much t can be delayed by t' through interferences happening on bank b, then we need a burst description denoted $fine(t, b)$ which is also greater than the burst description of any other execution trace, but in a different order, defined below.

The new order between burst descriptions is denoted $\preceq$ and defined as follows: If $e^1, e^2 \in \mathbb{B}_\infty$ then we write $e^1 \preceq e^2$ if for every $1 \leq k < \infty$ we have:

$$\sum_{i=1}^{k} i \times e^2(i) \geq \sum_{i=1}^{k} i \times e^1(i)$$

This amounts to e^2 taking at least as many cycles to pass arbitration, but having these cycles grouped into smaller bursts. The set $\mathbb{B}_\infty$, endowed with the $\preceq$ partial order, is a lattice. We shall denote with $\cup$ the greatest lower bound operator of this lattice. Note that, given that $\mathbb{B}_M$ is naturally included in $\mathbb{B}_\infty$ for any positive integer M, the $\preceq$ order and the $\cup$ operator are also defined on $\mathbb{B}_M$.

Intuitively, the worst-case interference scenario involves the largest possible number of (small) packets of the side that is delayed (burst description maximal in the sense of $\preceq$) and the greatest packets on the side that delays (burst description maximal in the sense of $\leq$).

6.2 L2 interferences

Under these definitions, assuming that two tasks t_1 and t_2 are executed on different processors and that $overlap(t_1, t_2) = true$, we can compute an upper bound on the overhead that t_2 can add to the execution time of t_1 due to interferences on bank b, which we denote $interfL2(t_1, t_2, b)$.

To do so, we assume that $coarse(t_2, b)$ and $fine(t_1, b)$ are provided and that $c = frag_n(coarse(t_2, b))$ and $f = frag_n(fine(t_1, b))$. We will also denote with $sum(f) = \sum_{k=1}^{n+1} f(k)$ the number of bursts (of any size) in f. Then:

$$interfL2(t_1, t_2, b) = \sum_{k=n+1}^{1} \left\{ k * min \left[c(k), max \left(0, sum(f) - \sum_{j=n+1}^{k+1} c(j) \right) \right] \right\} \quad (1)$$

Intuitively, for each burst in f, incrementally, we assume delaying by the largest remaining burst of c.

Starting from this basic brick involving only two tasks on a single SMEM bank, we can first determine an upper bound on the overhead that t_2 can add to the execution time of t_1 on any bank:

$$interfL2(t_1, t_2) = \sum_{k=0}^{15} interfL2(t_1, t_2, b_k)$$

Then, we can build a first over-approximation of the delay incurred by one task due to L2 interferences:

$$interfL2^0(t) = \sum_{overlap(t,t')=true} interfL2(t, t')$$

Of course, this last bound can usually be significantly improved. For instance, using the notations used in the definition of $interfL2(t_1, t_2, b)$, interferences coming from all the tasks executing on one core cannot comprise more than $sum(f)$ bursts (whereas by summing the individual contributions $interfL2(t, t')$ of all tasks t' running on a core this bound can be overflowed). To avoid this source of over-approximation, we can consider all interferences from one core p at once, and set:

$$c_{p,b} = \sum_{\substack{overlap(t,t')=true \\ p(t')=p}} frag_n(coarse(t', b)) \quad (2)$$

$$interfL2(t, p, b) = \sum_{k=n+1}^{1} \left\{ k * min \left[c_{p,b}(k), max \left(0, sum(f) - \sum_{j=n+1}^{k+1} c_{p,b}(j) \right) \right] \right\} \quad (3)$$

$$interfL2^1(t) = \sum_{p \neq p(t)} \sum_{k=0}^{15} interfL2(t, p, b_k) \quad (4)$$

Similarly, considering more elaborate task models (*e.g.* DAGs) or time/space isolation properties (discussed in Section 8) should allow further reducing interferences.

7 Experimental results

In this section, we evaluate our interference analysis methods on a number of applications chosen for their representativity. One particular point of this evaluation is that no static analysis tools, such as WCET analysis tools, exist yet for the Kalray MPPA3 processor (only for its predecessor MPPA2). For this reason, we have had to derive the input of our algorithms (the burst descriptions) from mere execution traces of the applications. The process, detailed below, uses simplified versions of the method in Section 5.5. The result is not meant to be safe, but simply to give a quantitative evaluation of our interference analysis method that is as realistic as possible.

7.1 The testbench

We have applied our analysis methods on 7 tasks taken from 4 applications. The basic characteristics of these applications are summarized in Table 1. To obtain these figures, the applications have been compiled under maximum optimization (`gcc -O3`) and have been executed once (in a

single configuration) to obtain an execution trace.[21] The figures in the four columns correspond to:

- ReadIC - Number of instruction cache misses. Each of them results in two collated requests from the instruction cache to the memory.
- ReadDC - Number of data cache misses. Each if them results in two collated requests from the data cache to the memory.
- Write - Data writes, each resulting in a single request to the memory.
- Execution time - total execution time in isolation.

All applications feature a fully static memory allocation (no use of `malloc`).

Application source	Task description	Memory access profile			Execution time (cycles)
		ReadIC	ReadDC	W	
Industrial avionics engine control (SCADE-generated)	task1	733	446	2920	20479
	task2	703	306	2247	16481
	task3	451	277	1511	11374
	task4	395	183	1353	9797
TACLE bench[8]	Anagram function	84	588	84591	726069
	Pattern Matching (PM)	180	296	210890	4233766
Linear algebra/ML	Matrix multiplication (matmul)	5	384	32768	251987

Table 1: Application and task characteristics

Four of the seven tasks are extracted from a large, real-life avionics engine control application. Following the industrial process, the application is specified in SCADE and automatically translated into C. We have worked directly on the generated C code. As the figures in Table 1 show, these tasks are control-dominated, with many more instruction memory reads than data memory reads. This is typical for SCADE-generated code of embedded control applications.

The linear algebra code (a 32x32x32 floating point matrix multiplication) is the exact opposite: the number of instruction memory reads is far smaller than data memory reads. This is typical for high-performance code in both AI/ML and model-predictive control.

The two remaining applications have been selected from the TACLe WCET benchmark suite [8] for their intermediate memory access profile.

7.2 Extraction of burst descriptions

As explained above, in the absence of static analysis tools, the input data needed by our interference analysis is extracted from execution traces. Code is executed in the cycle-accurate MPPA3 processor simulator provided by Kalray. The resulting execution trace is then passed through an LRU cache simulator we developed. This pass determines which data load instructions result in data cache misses (an thus in data memory read requests). It also determines an upper bound on the number of instruction memory requests. This information is annotated in the traces.

[21] Maximum optimization greatly reduces the number of memory access operations, thus reducing the absolute numbers of interferences. It also packs instructions into bundles, which increases the relative weight of each interference as a percentage of execution time.

From the annotated trace of each task t we extract replacements for the following pieces of data used in the formulas of Sections 5.4 and 6.2:

- $e^s(t)$ is a replacement for $e(t)$, needed in the computation of the L1 interferences (and originally computed in Section 5.5).
- $c(t,b)$ is a replacement for $frag_n(coarse(t,b))$, needed in the computation of L2 interferences, in equations 1 and 2.
- $s(t,b)$ is a replacement for $sum(frag_n(fine(t,b)))$, needed in the computation of L2 interferences, in equations 1 and 3.

The computation of the values needed at L2 level is only performed for $n = 1$. Without taking into account tight inter-processor synchronizations to synchronize bursts coming from the different PEs (which are not considered by the algorithms we define and are difficult to enforce) we expect interference upper bounds to worsen when n increases, as $frag_n(coarse(t,b))$ will increase faster than $frag_n(fine(t,b))$.

Computation of $e^s(t)$ It is performed using a modified version of the recurrence relations of Section 5.5. The modifications are the following:

- Whenever the original recurrence equations consider a supremum over multiple possible histories, we only consider those compatible with our execution trace:
  - The set $src(p)$ used in cases 2 and 4 contains exactly one element.
  - The set $P(p)$ used in case 3 only contains subsequences of bundles of the trace ending in the current bundle.
- Instead of May or Must data cache analysis results, we use exact cache states.

Computation of $c(t,b)$ It is performed by counting the number m of memory requests to bank b, and then dividing them into a minimum number of bursts:

$$c(t,b) = \begin{cases} (n+1 \to m \ div \ (n+1)) & \text{if } m \ mod \ (n+1) = 0 \\ (n+1 \to m \ div \ (n+1); m \ mod \ (n+1) \to 1) & \text{otherwise} \end{cases}$$

Computation of $s(t,b)$ This over-approximation is less brutal. We start by identifying the number i of instruction read requests towards bank b. Each of these are considered as a single burst of size 2. Then, we identify the bursts of data accesses (sequences of data accesses that reach bank b in successive cycles). These are individually cut into a minimal number of bursts of size at most $n+1$ and counted into $s(t,b)$.

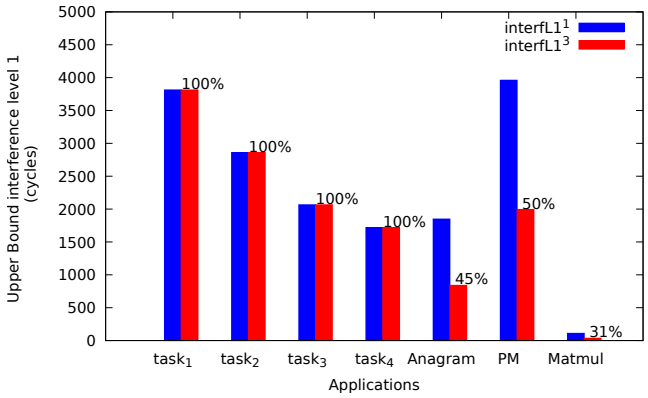
7.3 Results

7.4 L1 interferences

The result of applying L1 interference analysis is provided in Fig. 3. The graph at left compares our best analysis ($interfL1^3$) to the $interfL1^1$ analysis. The figures are provided both as absolute values and our analysis as a percentage of $interfL1^1$. The table at right evaluates $interfL1^3$ as a percentage of the task execution time.

We use $interfL1^1$ as a baseline instead of the $interfL1^0$ analysis introduced in previous work [10] for several reasons: First of all, $interfL1^0$ is by construction less tight than $interfL1^1$.

Second, in the case of avionics tasks the pessimism of $interfL1^0$ is too significant, and easily corrected. For instance, $interfL1^1(task1) = 3812$, whereas $interfL1^0(task1) = 15393$, or 75% of the execution time of task1. The final reason is that using $interfL1^0$ as a baseline would have downplayed the importance of the application profiles identified below.



Task	L1 interference	Exec. time
task1	18.62%	20472
task2	17.34%	16481
task3	18.15%	11373
task4	17.56%	9788
matmul	$1.38 * 10^{-4}$%	251987
anagram	$1.15 * 10^{-3}$%	726280
pm	$4.70 * 10^{-4}$%	4233766

Figure 3: L1 interference analysis results. At left, precision improvement ($interfL1^1$ in blue vs. $interfL1^3$ in red). At right, $interfL1^3$ as a percentage of execution time.

One immediately notices two very different application profiles: the control-dominated tasks of the avionics application vs. all the other tasks. In the first case, worst-case interferences as a percentage of WCET are very large, and our method provides no reduction in interference upper bound. To understand the reason for this, recall that interferences are due to instruction requests which are delayed by bursts of data requests. Our analysis determines that bursts of data requests are significantly smaller than the worst case d. This significantly reduces the interference upper bound when the number of bursts is smaller than the number of instruction requests (as is the case in tasks matmul, anagram, and PM, with reductions of 50%-70%). But this is not the case in the avionics tasks. For such control-dominated code the best method for reducing interferences is to reduce the number of data memory accesses (and in particular data writes, as explained in Section 8). An important aspect to remember here is that L1 interference cost is incurred only once per task (meaning that the figures are likely acceptable in practice).

7.5 L2 interferences

To evaluate L2 interferences, we have considered scenarios where one task is delayed through interferences by exactly one other task (including a copy if itself). This makes for a total of 49 cases. In each of them, we have considered two configurations corresponding to different memory allocation choices.

C1 Global data accessed by both tasks is placed on the same memory bank, but the stack and code of the two tasks are placed on different banks. This case is representative of a configuration where interferences are only due to shared variables and to accesses to system routines, such as `memcpy`, which can be accessed by both tasks.

C2 Accesses of the delayed task to global data, stack, and code can all be subject to interferences from accesses of the delayer task to global variables (but not code and stack). This case is meant to represent configurations where each processor is assigned one memory bank

which contains its stack, the code of tasks executing on it, and part of the shared data. In this case, interferences come under the form of reads and writes of the shared variables. We consider in this case, too, interferences due to the access to system routines.

The results of the analysis in these two cases are provided in Figures 2 and 3.

Delayed task	Delayer task							Delayed task exec. time
	task1	task2	task3	task4	matmul	anagram	PM	
task1	1.79	0.76	1.09	0.64	0.37	0.47	0.41	20472
task2	1.04	0.94	0.94	0.80	0.15	0.29	0.20	16481
task3	2.03	1.37	1.97	1.16	0.14	0.33	0.21	11373
task4	1.43	1.39	1.33	1.34	0.18	0.40	0.26	9788
matmul	0.01	0.005	0.003	0.003	13.30	5.50	26.31	251987
anagram	0.008	0.005	0.004	0.004	3.07	1.91	3.07	726280
PM	0.001	0.0005	0.0004	0.0004	0.79	0.32	4.69	4233766

Table 2: L2 analysis results in configuration C1. Figures are in % of execution time of the delayed task (which is provided for reference).

Delayed task	Delayer task							Delayed task exec. time
	task1	task2	task3	task4	matmul	anagram	PM	
task1	1.79	0.76	1.09	0.64	10.28	10.39	10.32	20472
task2	1.12	0.94	0.94	0.80	11.30	11.44	11.35	16481
task3	2.24	1.37	1.97	1.16	11.04	11.23	11.11	11373
task4	1.64	1.39	1.33	1.34	10.72	10.95	10.80	9788
matmul	0.015	0.005	0.003	0.003	13.30	5.50	26.31	251987
anagram	0.008	0.005	0.004	0.004	3.11	1.91	3.11	726280
PM	0.001	0.0005	0.0004	0.0004	0.79	0.32	4.69	4233766

Table 3: L2 analysis results in configuration C2

The most important remark is that for all but the most memory-intensive applications (matmul and PM) L2 interferences remain low (under 2%) when remaining in the same application class. This allows considering realistic situations where multiple interference sources must be considered for each task (but the number of interferences is kept under control through the mapping [7]).

In the second scenario, we can see that tasks generating large numbers of accesses to shared data (matmul, anagram, PM) can disproportionately delay smaller tasks if code and stack accesses are considered. This suggests that mapping of such tasks must be performed with care, drastically limiting interferences by means of time/space isolation, as discussed in the next section.

However, isolation may pose a problem for HPC code like that of matmul (which has the largest interference figures) because such applications often involve parallelized loop nests involving multiple processors accessing the same data. More work is needed to determine how the implementation of such applications can be performed in a way that reconciles performance and predictability.

8 Reducing interferences by code transformations

Interference analysis improvements are still possible, but significant interference reductions can be attained by transforming the application code.

8.1 Increasing memory bandwidth usage

Note in Table 1 that the number of write operations dwarfs the number of reads (both instruction and data), by a ratio of up to 440 times (for PM). This is of course a by-product of the write-through cache policy, but it is also exacerbated by the systematic under-use of the local interconnect width during write operations. To understand how serious an issue this is, we provide in Table 4 a classification of the write operations issued by two tasks according to the amount of the data that is stored. While each write operation can transport up to 32 bytes, 3488 of the memory accesses of task PM only transport 1 byte. Overall, PM uses only 15% of the bandwidth provided by its 210890 write operations, while task1 goes up to 27% due to the use of larger data types.

Task	Store size (bytes)						Bandwidth usage (%)
	1	2	4	8	16	32	
PM	3488	120	155034	51989	136	123	15%
task1	2	0	1641	196	1061	20	27%

Table 4: Number of store operations by store size in two applications

These figures suggest that in the PM task the number of memory write accesses could be divided by up to 6.5, and for task1 by 3. However, these are theoretical upper bounds, and we wanted to have a more realistic evaluation of potential gains. For a (very partial) result, we considered a simple optimization of the `memcpy` library routine. Calls to `memcpy` are generated by `gcc` to encode the copy of large C `struct` objects (in the wrappers calling SCADE-generated code). The stock implementation of `memcpy` on MPPA3 uses 16-byte load and store instructions instead of the full-bandwidth 32-byte instructions. By simply optimizing the `memcpy` routine to use full-bandwidth memory accesses, we reduce interferences as pictured in Fig. 4. Considering the store operation profiles of Table 4, we can determine that most of the 16-byte stores of task1 will be replaced with 32-byte stores, leading to a reduction of up to 530 memory accesses, or 16% (which is well approached by the 13% reduction in the task1 interferences of Fig. 4).

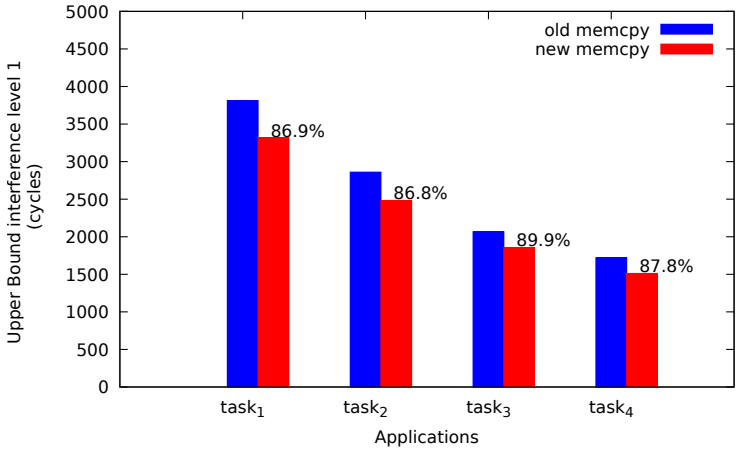


Figure 4: L1 interference reduction after `memcpy` optimization

Beyond manual code rewriting (as done for `memcpy`), increasing memory bandwidth use (and thus reducing the number of memory requests and the interferences) can only be done using non-trivial compiler support.[22]

8.2 Time/space isolation

Various isolation properties [2] can be used to reduce or completely eliminate L2 interferences. In turn, this also helps L1 analysis, by reducing the variability of the FP arbitration[23]

However, isolation has its cost [2], which can be unacceptable, especially in the case of parallelized HPC code (AI/ML, digital twin models, *etc.*). More research is therefore needed in this direction to determine the acceptable compromises.

9 Conclusion

This paper brings two main contributions. The first is a precise analysis of the properties of three novel architectural innovations of the MPPA3 architecture: the speculative PFB, the FP L1 arbiters, and the SAP L2 arbiters. This analysis both emphasizes the importance of memory access burstiness and allows defining methods to reduce interferences by construction. The second contribution is a formal model allowing the representation and worst-case reasoning on bursty memory access pattern specifications. This model supports the definition of novel algorithms for L1 and L2 memory access interference analysis, which provide better results than previous ones. We evaluate our model and algorithms on multiple applications.

Our analysis and results suggest that, to allow the safe and efficient mapping of high-performance applications, three complementary research directions should be followed in the future:

[22] The current MPPA3 port of `gcc` is limited in this respect.

[23] For instance, absence of L2 interferences means that store requests are never grouped into larger bursts due to waiting at the level of the FP arbiter. This reduces L1 interference analysis complexity and potentially reduces L1 interferences.

- The design of static analysis methods capable to extract the information required by our interference analysis, as well as improving our interference analysis methods.
- The introduction of application restructuring methods to increase memory bandwidth usage.
- The definition of parallelization methods allowing to enforce time/space isolation properties that reduce interferences without penalizing performance.

References

[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Proceedings SEUS*, 2010.

[2] P. Baufreton, V. Bregeon, K. Didier, G. Iooss, D. Potop-Butucaru, and J. Souyris. Efficient fine-grain parallelism in shared memory for real-time avionics. In *Proceedings ERTS2*, 2020.

[3] Thomas Carle, Manel Djemal, Daniela Genius, François Pêcheux, Dumitru Potop-Butucaru, Robert De Simone, Franck Wajsbürt, and Zhen Zhang. Reconciling performance and predictability on a many-core through off-line mapping. In *9th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC'2014)*, Montpellier, France, May 2014.

[4] R. Davis, S. Altmeyer, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. An extensible framework for multicore response time analysis. *Real-Time Systems*, 54(3):607–661, 2018.

[5] B. Dupont de Dinechin. Engineering a manycore processor platform for mission-critical applications. Keynote talk at MCSoC, 2016. `https://mcsoc-forum.org/2016/wp-content/uploads/2015/07/Keynote-De-Dinechin.pdf`.

[6] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire MaÃ¯za. Scaling up the memory interference analysis for hard real-time many-core systems. In *Design, Automation and Test in Europe Conference (DATE)*, 2020.

[7] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton, and A. Graillat. Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware. *ACM TACO*, 16(3):1–27, August 2019.

[8] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. Bo Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proceedings WCET*, 2016.

[9] Kalray. *Kalray kv3 VLIW Core Architecture Manual*, 2019. Manual KETD-62.

[10] Kalray. Time-critical computing on the mppa processor. Internal report KETD-417, 2021.

[11] H. Ozaktas, C. Rochange, and P. Sainrat. Automatic WCET analysis of real-time parallel applications. In *Proceedings of the WCET workshop*, Paris, France, 2013.

[12] Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet. Mapping hard real-time applications on many-core processors. In *Real-Time Networks and Systems (RTNS2016)*, Brest, France, October 2016.

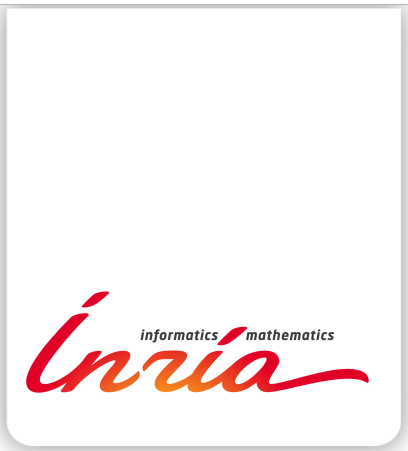
[13] D. Potop-Butucaru and I. Puaut. Integrated worst-case execution time estimation of multicore applications. In *Proceedings WCET, Paris, France*, 2013.

[14] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *Proceedings WCET*, 2006.

[15] H. Rihani, M. Moy, C. Maiza, R. Davis, and S. Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In *Proceedings RTNS*, 2016.

[16] B. Rouxel, , S. Derrien, and I. Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1 – 20, October 2017.

[17] S. Skalistis and A. Simalatsar. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *Proceedings DATE*, 2017.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399